This lecture covers the basic structure of F# programs, and some basic types of values: integers, strings, booleans, pairs.
Program structure

- An F# program generally consists of a number of declarations.
  - Later we’ll see declarations of types, exceptions, etc.
  - We’ll start with the most common kind: let declarations

- let declarations define names for values
  ```
  let numStudents = 31
  ```

- After the = we can have any expression: it will be evaluated to a value and used as the definition for the name
  ```
  let numStaff = (2 + 6) / 2
  ```

- Values are expressions that are already fully evaluated.
- F# values include numbers, strings, Booleans and more.
- Values even include functions, written using fun:
  ```
  let square = fun x -> x * x
  ```

- Unlike many languages, this means functions and non-functions can be defined in the same way.
Functions

- Having functions as a kind of values is powerful, as we’ll see later.
  - In fact, this is one of the defining features of functional programming.
- Since function definitions are common, they can be abbreviated:
  
  ```
  let square x = x * x               // Abbreviation
  let square = fun x -> x * x       // Without the abbreviation
  ```

- To call (or “apply”) a function, we just put it in front of an expression:
  
  ```
  let mySqrA = square 5
  let mySqrB = square (square (2+5))
  let mySqrC = (fun x -> x * x) 5
  ```

- Note that unlike many languages, parentheses are not needed in function definitions/values nor in applications.
  - Instead parens are used only when needed to group things together.
  - When we come to complicated uses of functions, this will be important, since otherwise there would be too many parentheses.
Functions, indentation and the top-level

- A `let` declaration may contain other declarations:

```
let doubleSqr x =
  let y = square x
  y + y          // Returned by doubleSqr
```

- Here the body of `doubleSqr` (after the `=`) is a `let`-expression that evaluates “`square x`”, calls this result `y`, then evaluates `y + y`.

- Indentation is important in F#! (Like Python, Haskell)
- The body of a `let` must be indented further than the `let`.
- Instead of closing braces, un-indentation tells the compiler where a declaration ends.
- You’ll need to put `#light` at the top of your program, except in the most recent version of F#.

- As we’ll see in the labs, F# has an interactive top-level that is often used for experimentation and debugging.
  - This allows you to enter expressions and declarations you want to be evaluated, and these can use the declarations in your program.
Types and functions

- Every variable and expression in F# must have a type assigned to it.
  - The compiler infers these types automatically and reports inconsistent types as errors.
  - Inference means that you usually do not need to declare the types for variables.
  - But, it is sometimes useful to add types to debug type errors, for readability, and to help the inference:
    
    ```
    let doubleSqr (x : int) = square x + square x
    ```

- Functions are given types like: `int -> int`
  - And, (again) functions are just another kind of value.
  - We’ll come back to this in the next topic.
Basic types: numbers

- To write interesting functions, we need interesting types for the inputs and outputs.
- F# provides some built-in types, and the programmer can also declare their own. (later)
- int is a built-in type, with standard operations like + - * / %.
- Similarly, there is float, float32, uint16 (unsigned 16 bit), nativeint (matches CPU), bigint (arbitrarily large)...
  - Functions like + - * / % are overloaded meaning that they can be used for many different types.
  - Unlike Java, C, etc., conversions are not done automatically between numeric types – instead there are functions. E.g.

```fsharp
let myAdd (i : integer) = (float i) + 0.5
```

- A float must include a decimal point: 1.0 (not 1)
- We will see more details in the labs.
Basic types: strings

- **string** is a built-in type, with + for concatenation.
  - Strings can be accessed like arrays.
  - But they are immutable – they can’t be modified.

- The following shows three examples, as a session with the F# interactive top-level:

  ```fsharp
  > let s = "CITS3242";;
  val s : string
  
  > s.Length;;
  val it : int = 8
  
  > s.[4];;
  val it : char = ‘3’
  ```

  [Here: the > is the prompt, ;; is required to end each piece of code, and the lines beginning with val are the results.]
Basic types: bool and unit

- **bool** is a built-in type with two values: true and false.
  - There are standard functions like &&, ||, <, >
  - <> is used for “not equal”. (Unlike Java, C.)

- **unit** is a built-in type with one value:
  ()
  - It is used when a function has no interesting input or output.
  - It is mostly used when some imperative effects are involved. (which we’ll see later)
Structured types: tuples

- F# also includes a number of ways of constructing types from simpler ones – the first one we will see is tuples.

- Tuples are just like cartesian products in mathematics: a compound value is built from a number of smaller values.

- `int * int` is a type for pairs (2-tuples) of integers
  - `(32, 42)` is a value in this type.

- `int * float * bool` is a type for triples (3-tuples)
  - `(32, 42.0, true)` is a value in this type.

- The parts of a tuple are usually accessed by deconstructing the tuple via pattern matching. E.g.:
  - ```
    let sumSqr (x,y) = square x + square y
  ```
  - This function takes one argument, a pair – but also similar to having 2 args.
  - Thus, it’s similar to functions/methods with multiple arguments in C/Java
  - But, pairs can do more: returning multiple values, put pairs into data structures, etc., very easily. (Compare to C/Java.)
Pattern matching

- Pattern matching can also be used with booleans, numbers, and other types. (We’ll see lists next time.)

- `match` expressions allow us to match against multiple patterns
  - The first pattern that matches the value is chosen.
  - Any variables in the pattern are `bound` to the corresponding parts of the value being matched.

    ```
    let xor boolPair = // xor : bool * bool -> bool
        match boolPair with
        | (false, x) -> x
        | (true, y) -> not y
    ```

- Functions that match arguments are common - they can be abbreviated as:

    ```
    let xor = function // “function” allows many patterns, unlike “fun”
        | (false, x) -> x // (but “fun” allows many arguments)
        | (true, y) -> not y
    ```

- Patterns deconstruct values in a way that mirrors the way they were constructed.

- Pattern matching is thus a very natural way of deconstructing (taking apart) values and acting based on their structure.